

AD-777 956

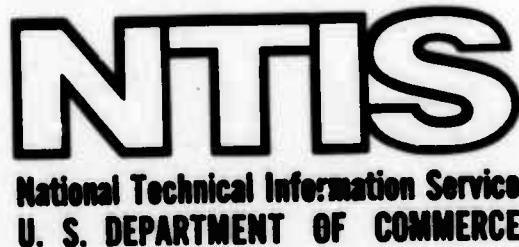
DOMAIN-INDEPENDENT AUTOMATIC PROGRAMMING

UNIVERSITY OF SOUTHERN CALIFORNIA

PREPARED FOR
ADVANCED RESEARCH PROJECTS AGENCY

MARCH 1974

DISTRIBUTED BY:



REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/RR-73-14	2. GVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD-777-956
4. TITLE (and Subtitle) Domain-Independent Automatic Programming		5. TYPE OF REPORT & PERIOD COVERED Research report
7. AUTHOR(s) Robert M. Balzer, Norton R. Greenfeld, Martin J. Kay, William C. Mann, Walter R. Ryder, David Wilczynski, Albert L. Zobrist		8. CONTRACT OR GRANT NUMBER(s) DAHC 15 72 C0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, California 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order #2223/1
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, Virginia 22209		12. REPORT DATE March 1974
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) -----		13. NUMBER OF PAGES 24
16. DISTRIBUTION STATEMENT (of this Report) Distribution unlimited. Available from National Technical Information Service, Springfield, Virginia 22151		15. SECURITY CLASS. (of this report) none
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) -----		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
18. SUPPLEMENTARY NOTES To be presented at IFIP Congress 74, 5-10 August 1974, Stockholm, Sweden. To be published in <u>Proceedings of IFIP Congress 74.</u>		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Automatic programming, domain-independent, model acquisition, natural language, nonprocedural languages, nonprofessional computer users, problem specification, process transformation.		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) PAGE V		

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
U S Department of Commerce
Springfield VA 22151

ABSTRACT

An automatic programming system is distinguished from a conventional programming system by its use of an explicit semantic model of the application domain to structure the dialogue between the system and the user, to understand the user's responses, and to translate these into actions. The major differences between the design effort reported here (and the project's main focuses) and other automatic programming projects are: first, its independence of any particular domain and its dialogue-driven acquisition of the domain to produce a Loose Model; second, the informal and typically ill-structured manner in which both this Loose Model and the task to be programmed are specified and their translation into a directly interpretable Precise Model.

Throughout the system, knowledge is represented by tuples, and structured by: a theory of domains and their model interrelationships; a strong notion of types; and the use of constraints on all arguments of the tuples. Use of compound expressions, enabling the intermixing of patterns to be instantiated with expressions to be evaluated, greatly simplifies program control structure. The system also enables constraints and inferences, as well as actions, to be represented as procedures, which can be used in both a goal-directed and applicative manner. A detailed example illustrates these capabilities.

The research, sponsored by ARPA under Contract No. DAHC15 72 C 0308, ARPA Order No. 2223/1, Program Code No. 3D30 and 3P10, is directed toward vast improvement in both efficiency and quality of the production of software. The work is of particular importance to the large very diverse application software packages being developed by all branches of the Military.



Robert M. Balzer
Norton R. Greenfeld
Martin J. Kay
William C. Mann
Walter R. Ryder
David Wilczynski
Albert L. Zobrist

ISI/RR-73-14
March 1974

Domain-Independent Automatic Programming

UNIVERSITY OF SOUTHERN CALIFORNIA



INFORMATION SCIENCES INSTITUTE

4676 Admiralty Way / Marina del Rey / California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308. ARPA ORDER NO. 2223/1, PROGRAM CODE NO. 3D30 AND 3P10.

VIEWS AND CONCLUSIONS CONTAINED IN THIS STUDY ARE THE AUTHOR'S AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL OPINION OR POLICY OF THE UNIVERSITY OF SOUTHERN CALIFORNIA OR ANY OTHER PERSON OR AGENCY CONNECTED WITH IT.

DOCUMENT APPROVED FOR PUBLIC RELEASE AND SALE: DISTRIBUTION IS UNLIMITED.

CONTENTS

Abstract	v
Acknowledgments	vi
Introduction	1
Overall System Structure	3
Knowledge Representation	5
Domain Acquisition	7
Model Completion	8
Precise Model	10
Example	13
Conclusion	17
References	18

ABSTRACT

An automatic programming system is distinguished from a conventional programming system by its use of an explicit semantic model of the application domain to structure the dialogue between the system and the user, to understand the user's responses, and to translate these into actions. The major differences between the design effort reported here (and the project's main focuses) and other automatic programming projects are: first, its independence of any particular domain and its dialogue-driven acquisition of the domain to produce a Loose Model; second, the informal and typically ill-structured manner in which both this Loose Model and the task to be programmed are specified and their translation into a directly interpretable Precise Model.

Throughout the system, knowledge is represented by tuples, and structured by: a theory of domains and their model interrelationships; a strong notion of types; and the use of constraints on all arguments of the tuples. Use of compound expressions, enabling the intermixing of patterns to be instantiated with expressions to be evaluated, greatly simplifies program control structure. The system also enables constraints and inferences, as well as actions, to be represented as procedures, which can be used in both a goal-directed and applicative manner. A detailed example illustrates these capabilities.

The research, sponsored by ARPA under Contract No. DAHC15 72 C 0308, ARPA Order No. 2223/1, Program Code No. 3D30 and 3P10, is directed toward vast improvement in both efficiency and quality of the production of software. The work is of particular importance to the large very diverse application software packages being developed by all branches of the Military.

ACKNOWLEDGMENTS

This is a group project in the true sense, and we gratefully acknowledge the contributions to our discussions of Richard Hale, Nadine Malcolm, Robert Lingard, and our summer visitors, Bill Mark, Andee Rubin, and Beau Shell. We also wish to acknowledge the role played by the Institute for the Future's FORUM teleconferencing system in recording and communicating our ideas with each other.

INTRODUCTION

The work reported here represents the first phase of the automatic programming project at the USC/Information Sciences Institute (ISI). After an initial survey of current work in the field, the group developed a plan for attacking what appeared to be the fundamental issues. This took the form of an actual system, the design and implementation of which is now in its early stages. Rather than enter into a detailed discussion of what we understand automatic programming to be, it will emerge from the discussion of the system being built. The results of the initial survey and the overall view of the field adopted are reported elsewhere[1,2]. One point of that view should be stressed here. This project is not seen as an incremental advance in computer languages or the art of programming, but rather as an attempt to make the power of the computer available to a large class of users without the necessity of a step similar to the one now called programming. Ultimately, a client should be able to negotiate directly with a computer system in much the same terms as he now negotiates with a programmer.

Computer usage generally falls into two categories: use of existing programs or creation of new ones. There is no sharp distinction between the two because data fed into existing programs can be thought of as instructions which program their behavior, and because the creation of new programs utilizes either compilers or interpreters which treat such instructions as data. Also, the techniques for translating a task into appropriate input for the two are very similar. Nevertheless, we have chosen to deal only with programming activities, which we regard as the process of translating a task to be performed into a computer language, taking into account the constraints and limitations of both the computer and the domain of interest from which the task was drawn.

The constraints and restrictions of the computer have increasingly been incorporated and internalized in programming advances for several years. They are manifest in better languages, automatic storage mechanisms, and optimizations of many forms.

On the other hand, the structure, constraints, and limitations of the problem domain have generally not been incorporated into programming systems. The utilization of such knowledge is a major theme of automatic programming that characterizes the distinction between it and conventional programming, and raises a number of issues. If the system is to understand something of a domain -- a particular universe of discourse -- how is the knowledge on which this understanding is based to be represented? What procedures can be made available for exploiting this knowledge in guiding the system's interaction with a user and in generating programs? How, in particular, is the essentially nonprocedural information in constraints and limitations to be reflected in a procedural form? What can be done to help identify inconsistencies? How can the system be given a capacity for inference similar to the one that forms the mainstay of human communication and which allows obvious details to be left unspecified? Will the system be able to understand its own products well enough to be able to modify them in response to changed requirements? Answers to these questions define the front on which important advances in automatic programming will be made.

Hitherto, the designers of programming systems have concentrated their attention on creating an instrument that would be easy to play. Like all instruments, the system had a purely passive role in the programming enterprise. We, on the other hand, took the view that the problem of programming is largely a problem of communication and that communication, to be easy and natural, must be with an active agent.

Thus the main distinction between conventional and automatic programming is the latter's use of a semantic model of a domain to structure the dialogue between the system and the user, to understand the user's responses, and to translate the user's responses into actions. The major distinctions between the work reported here and other automatic programming efforts are: first, its independence of any particular domain and its acquisition of the domain model through a dialogue with the user; second, the informal and typically ill-structured manner in which both the domain semantics and the task to be programmed are specified. In fact, these two areas represent the two main focuses of the project: dialogue-driven acquisition of a domain and translation of ill-defined specifications into a precise form.

OVERALL SYSTEM STRUCTURE

In our plan, the automatic programming system consists of four processing modules and six data bases. The data bases consist, as much as possible, of descriptive (rather than imperative) knowledge, organized so that the system can use this knowledge in many different ways. These data bases have been segregated because of the different logical functions they perform and because of the way they are treated by the different processing modules.

Data Bases

The Domain Knowledge data base contains all the descriptive information about the problem domain, such as the types of objects which can exist in the domain and their descriptions, the types of actions which can occur in the domain, the relations which may exist between objects or events (action occurrences), and any constraints which must be satisfied by the domain.

The Domain Model contains, at any point in time, an instantaneous snapshot of the instantiated objects in the domain and their relationship with other objects in the domain. It represents, through time, a direct simulation of the problem domain.

The Loose Model contains the problem statement in an imprecise form which may be incomplete or ambiguous and which can only be understood in the context of the information in the Domain Knowledge and Domain Model data bases.

The Precise Model, on the other hand, represents a precise, complete, unambiguous, and directly interpretable process for solving the posed problem.

The Strategy Knowledge data base consists of information which guides the choice of actions and/or objects for those actions when alternative possibilities exist within the domain.

Finally, the Script data base contains partially filled in forms which guide the dialogue between the system and the user and are dynamically altered on the basis of the user's input and by the demands of the Model Completion module.

Processing Modules

Initially, to simplify the implementation, the processing modules will be highly self-contained and have only a limited knowledge of the processing and requirements of other modules. Later these modules will be more highly integrated and cooperative.

The Domain Acquisition module is responsible for all communications with the user, with building the Domain Knowledge and Domain Model data bases, with obtaining the Loose Model statement, with determining on syntactic grounds the well-formedness of all this information, with building and modifying the Script, and with using it to direct the dialogue for the acquisition of further information necessary for such syntactic well-formedness or requested by the Model Completion module.

The Model Completion module takes the Loose Model and determines its semantic well-formedness on the basis of the information in the Domain Knowledge and Domain Model data bases. It is responsible for transforming the Loose Model into an operational interpretable form called the Precise Model. Any inability to perform this transformation results in a description of the cause being passed back through the Script to the Domain Acquisition phase which then interacts with the user to correct the deficiency (usually by adding more knowledge about the domain to the Domain Knowledge data base).

The Interpreter executes the action sequences in the Precise Model and updates the Domain Model accordingly. It is responsible for locating objects defined descriptively, for evaluating conditions to select alternative sequences of actions, and for maintaining restrictions on domain behavior.

The Data Base Handler is responsible for maintaining the various data bases, deciding on store-recompute policy, maintaining consistency, and (through inference) obscuring the difference between explicit and implicit data.

A primary objective of our project has been the creation of a core experimental system for testing progress on Domain Acquisition and Model Completion. As such, the Interpreter and Data Base Handler have been completely specified, and will be used for both the Precise Model and the implementation of the automatic programming system itself. To fully utilize these implementation capabilities, the Domain Acquisition and Model Completion modules will be treated as domains with their own actions, objects, constraints, and rules of inference. This bootstrapping will focus attention on the real problems of using our approach in complex domains.

A more detailed description of the system is given in the following sections by focusing on the major components: the representation of knowledge, the transformation performed by the Domain Acquisition and Model Completion modules, and the form of the Precise Model produced by Model Completion. This description is followed by an annotated example.

KNOWLEDGE REPRESENTATION

Throughout the system, knowledge is represented as stored tuples. The first element of any tuple specifies the type of tuple and the rest of the elements are the arguments for that tuple. Each stored tuple is associated with a particular domain. Data bases are compartmentalized into separate domains which form a lattice. Each domain is defined as A-KIND-OF (AKO) another domain and this structure forms the basis of the domain lattice. The interpretation of the lattice structure is that, unless specifically prohibited, properties (of all types) from higher level domains are inherited by lower level ones.

The structure of knowledge in the system is highly constrained by two mechanisms: types and constraints. Each element of a tuple must be of a type acceptable for that argument as specified in the definition of that kind of tuple. Like domains, types are defined by A-KIND-OF relation and form lattices. (This structure is very similar to MAPL[3].) An element of a tuple is acceptable if its type is the same as that specified in the tuple definition, or if its type is a lattice descendant of the specified type. In addition to type acceptability, the elements of a tuple must also satisfy arbitrary

constraints specified in the tuple definition. These constraints are checked at the time that the tuple is added to a domain.

A domain consists of types (objects), actions, relations, constraints, rules of inference, and instantiations of all of the above. Together with the type and constraint mechanisms for tuples, this knowledge of the kinds of information contained within a domain represents the syntactic basis used by the Domain Acquisition module to construct and modify its Script, and hence its dialogue with the user.

The following tuples are used to structure knowledge in the system:

(AKO x y z ...) --- x is a subclass (A-KIND-OF) of y. z is optional and if present is a further specification of y which ensures it is also an x. Following z can be optional arguments which specify a case frame for x consisting of properties which MAY, MUST (must exist), REQUIRED (must exist and also be known), or CANNOT be present, and CONSTRAINTS which must be satisfied by instances of an x. Any entity specified as an AKO is a TYPE in the system. OBJECTS, ACTIONS, CONSTRAINTS, INFERENCES, DOMAINS, and ATTRIBUTES can all be specified by AKO. In an AKO, y may be either a TYPE or an instantiation of a TYPE. Examples:

(AKO wife person (anu (sex * female) (marital-status * married)))
(AKO walk run (speed * slow))

The star (*) in the above further specifications refer to the entity being further specified. The first example defines a wife as a person whose sex is female and whose marital status is married.

(x y z ...) --- x is a RELATION or ACTION with y z ... as arguments. The arguments are ordered. The restrictions, names, and types of each argument can be found in a prototype description of x (in the A-RELATION-ON (ARO) or ACTION tuple).

(ARO x y z ... (CONSTRAINTS a b ...)) --- x is the name of ARO arguments y z ... specified by their name and allowed TYPE. The last element of the tuple may be a set of CONSTRAINTS which must be satisfied by an instantiation of the tuple. Any of the arguments can be named by specifying a pair (r s) where r is the name (used to identify a particular argument and to help the input system correctly position the arguments) and s is the TYPE.

(ACTION x y z ... r) --- x is the name of an ACTION. y z ... are the parameters of the action and use the same notation as ARO above. r is a set of attributes of the ACTION such as CONSTRAINTS to be satisfied by the parameters, who the ACTION is CONTROLLED-BY, a DESCRIPTION of the action, POSTCONDITIONS which can be asserted after it has been completed, PRECONDITIONS necessary before it is started, etc.

(INFER x y z) --- if y as a pattern can be instantiated then z can be ASSERTED. x is a list of variables to be bound in the patterns.

(CONSTRAINT x y) --- y is a pattern, or the negation of one, which must be satisfied before and after every ACTION defined in the associated DOMAIN (but not during those ACTIONS, and not before and after more primitive ACTIONS in another lower DOMAIN). x is a list of variables to be bound in the pattern.

DOMAIN ACQUISITION

The Domain Acquisition module has responsibility for communicating with the user in natural language and extracting from the dialogue the information needed to build the Domain Knowledge, Domain Model, and Loose Model data bases. The mechanism for guiding this dialogue is the Script. The basic idea is to use the regularities and restrictions in a domain to structure new knowledge about that domain and indicate where more information is required. Thus, each of the entities of a domain can be thought of in terms of an extended Case Grammar[4], which specifies a "frame" or form to be filled in for that entity. As with all forms, it has certain fields which must have specified types of information, others which may be present, absent, or present in varying amounts. It may also specify certain well-formedness criteria of a more global nature for entities of this type. The form represents a template which is to be instantiated in a domain.

These instantiated forms may be either fully or partially instantiated. Fully instantiated ones represent constants in the domain. Partially instantiated forms can be used both in building up the intrarelated structure of one of the data bases or as a form for further instantiation. In particular, such partially instantiated forms can be used in the Script as a guiding mechanism for the dialogue. In addition, some forms represent refinements of others which either fill in certain fields of that form or expand it by adding new fields which may or may not be filled in. Thus, forms can create either instantiated entities in a domain or further forms.

Such a structure suggests the development of a language for the description of forms and how they should be filled in. Domain Acquisition would then become a table-driven module which from its knowledge of communication (and natural language) and the particular form given it to fill in would engage in a dialogue with the user to obtain the necessary information. This view strengthens the conception of Domain Acquisition as the "syntactic" component, as it would not know what the fields in the form were, or how they were to be used, but only their syntactic construction and relationship to other fields in the form.

This conception has the advantage of focusing attention on how such a form could be used to direct the dialogue and would greatly simplify any changes in those forms necessitated by further understanding of Model Completion processing. It would also open the door for other Loose Models which might not be procedurally oriented. The problem with utilizing this technique is finding some way to capitalize on the regularity in a domain without imposing an undue rigidity on the dialogue or the forms of information accepted.

One technique being utilized to study the structuring and extraction of information from a dialogue is the analysis of dialogues in which the content words of a domain have been systematically replaced by nonsense words[5]. In these dialogues, one member of the group plays the role of the system while another plays that of a user. The analysis of these dialogues illustrates the difficulties encountered by an automatic programming system acquiring information in a new domain and is beginning to yield a set of applicable rules and techniques.

There are three components to the Domain Acquisition phase: a linguistic front end which translates natural language input into internal form; a dialectic component which utilizes the Script to guide the dialogue with the user; and a structure extraction and building component which uses tuple restrictions on element type and constraints to select the intended meaning of an input, spot inconsistencies, and determine the need for missing information. Work is centering on these last two components, leaving the linguistic front end for the future.

MODEL COMPLETION

The Loose Model represents an informal statement of a problem in a domain which can be processed with the aid of information contained in the Domain Model and Domain Knowledge data bases with the application of intelligence. Thus, with the right kind of data base access and processor, the Loose Model is interpretable. The main function of Model Completion is to reduce the intelligence requirements on the run-time processor and to limit the access during run-time to the Domain Knowledge data base. This distinction, though not sharp, lies at the very heart of programming. A program

embodies an algorithm and it is the essence of an algorithm that it does not KNOW what it is doing. In other words, it requires understanding of the problem domain to write a program, but the eventual program operates blindly. If a process must have recourse to an understanding of a domain to continue with the solution of a task, then it does not embody a method for solving that problem, and is therefore not a program. It is, instead, a problem solver which develops solutions essentially by (heuristic) trial and error. In programs, the need for such recourse has been anticipated and incorporated into the steps of the algorithm so that the structure of the domain and problem solving are no longer required during execution.

Such anticipation and removal of reliance on Domain Knowledge and problem solving can be regarded as a compiling process and is the main function of Model Completion. Closely related is the issue of efficiency which represents good ways of removing such dependencies. Our focus will be to produce running programs, not optimized ones. Hence, the concern is more with widening the range of transformations which can be performed on Loose Models and the freedoms thus allowed in the Loose Model specification than in eliminating redundant checks or optimally ordering the processing in the produced programs.

Thus Model Completion is the translator from the Loose to Precise Model. As such, its main responsibility is to transform actions into procedures. This involves filling in procedure invocations (fully instantiating the argument lists) and making these consistent with the procedure requirements; filling in missing links (making explicit the access path to required data); deciding explicitly when to perform bindings and evaluations in the Domain Model; deciding explicitly how to handle possible errors; identifying missing information and removing dependence on it until (and if) it is used during executions and performing back translations from Precise to Loose, both for describing execution behavior and for explaining why actions were selected. The annotated example following the Precise Model section illustrates the kinds of transformations planned for Model Completion.

One transformation particularly worth noting is the multiple use of actions in both the applicative and goal-directed forms. In Precise Model form, actions have a

set of parameters and local pattern match variables. It is assumed, upon entry to such an action, that the parameters have been bound and that the local variables are unbound. In a goal-directed invocation, as part of an ACHIEVE statement, an action is being invoked. It is invoked because its result matches a needed, but as yet unfulfilled, part of the form to be achieved. Since this occurs in the midst of a pattern-match, the form is partially instantiated and only some of the arguments needed for the action may have been determined already. Two possibilities for processing exist. The first is for the system to preselect possible values for the undetermined parameters, invoke the action, and if it fails try another set of values, and so on. The second possibility is that the action is modified (logically) so that the undetermined parameters are treated as local variables to be bound by the pattern matches within the action rather than by being determined from the outside. They, however, remain bound when the action is exited. Thus, conceptually, the undetermined arguments are bound by performing the action. This second possibility is much more reasonable, allowing the inherent constraints of the action to guide the bindings of the unbound arguments, and occurs automatically in the Precise Model. By definition, the pattern matcher instantiates all unbound variables encountered in a pattern and leaves unchanged those already bound. Hence any parameters which have a prespecified value upon entry to the routine will have that value unchanged, while those that are unbound will have an instantiated value assigned in the normal course of execution of the action. The binding mechanism in the Precise Model causes these instantiated values to automatically be reflected in the arguments of the invocation. A related issue is the possible bindings in the pattern-directed invocation of variables local to the invoked action. Unfortunately, such bindings are not automatically reflected in the invoked action and a special type of entry must be performed when such conditions arise.

PRECISE MODEL

The Precise Model is the restatement of the user's problem in the programming language AP/1[6]. This language is an extension of LISP[7], which supports associative relational data bases with the domain compartmentalization described earlier, strongly

typed variables, compound pattern matches, and failure control. Strong typing and compound patterns are especially important in simplifying the system's writing of the Precise Model by minimizing the translation between it and the Loose Model and by reducing and simplifying the control structures required. In fact, compound patterns have enabled backtracking to be completely eliminated and replaced by a single FOR loop which iterates through a set of instantiations of the compound pattern. It also enables intelligence to be applied, within the pattern matches, to determine how best to obtain valid instantiations.

Additionally, Model Completion utilizes only a subset of AP/I (which is also the implementation language for the project) to further simplify the writing and analysis of Precise Model programs. The major difference is that the Precise Model utilizes no free or local variables except for pattern match variables which are instantiated during the matching process. All communication between routines is either via explicit parameter passing or through data contained in the Domain Model.

AP/I generally allows the arbitrary mixing of tuples to be instantiated and functions to be evaluated. This includes the functions AND, OR, and NOT, as well as any other defined LISP functions. It is assumed that such functions have no side effects. Each tuple in an expression is treated as a function and evaluated if it has a function definition. If not, then it is treated as a pattern to be instantiated. Because there are no free variables, and the only local variables are pattern match variables, the rule for instantiation is very simple. Any parameter or variable which is unbound at the time it is encountered within a pattern is instantiated. Already bound variables are left unchanged.

The value of a pattern is always the instantiated version of that pattern if the match was successful or NIL otherwise. No other possibilities exist. Thus all pattern matches return either the instantiated pattern or NIL and the concept of failure does not exist within the pattern matcher. It always returns to its caller with one of these values.

The routines (statements) which invoke the pattern matcher may take other actions

with the returned value. They may extract from it particular bindings or subexpressions or cause failure when a NIL value is returned. Each of the "statements" in AP/I is, in fact, a function which uses the value returned from the pattern matcher as it sees fit. In this regard, the AND, OR, and NOT functions are no different than any other in the system.

One such useful function is Further Specification. It takes a typed variable and a pattern to be instantiated as its arguments. If the pattern is successfully instantiated, the value of the typed variable is returned as the value of the function and NIL is returned otherwise. Thus Further Specification can be viewed as "find the x such that <pattern>".

In AP/I the ATTEMPT statement is used to deal with all failures which occur in the attempted statement. The ATTEMPT statement also automatically creates a new context for the execution of the attempted statement. If the statement is successful, then the tuples in the context (which can be thought of as a temporary domain) are promoted to the context existing before the attempt. If not, all these tuples are removed from the system. Thus the side effects of failures are automatically removed from the system.

Any statement which can fail can have THEN and ELSE clauses attached to it. This includes the IS, ATTEMPT, ACHIEVE, ASSERT, REMOVE, FOR, and PERFORM statements. In each case, if the statement completes successfully, then the THEN clause, if present, is executed. Failure of the statement causes the execution of the ELSE clause which, if present, prevents further promulgation of the failure. The one exception to this is the ATTEMPT statement which handles failure whether or not an ELSE clause is present.

The FOR statements are used to loop through a set of instantiations of a pattern, either performing some operation on them, or searching for a single one which satisfies some criteria. The suspension and continuation of instantiations afforded by FOR statements is the only mechanism, outside the pattern matcher, for attempting a sequence of instantiations looking for a successful one. In this regard it is very CONNIVER-like[8], but it is only effective within the loop. There is no exit- and reentry-type capability. The pattern matcher has internal backtracking mechanisms for

searching for successful instantiations of patterns. The compound pattern matches are largely responsible for eliminating the need for backtracking in the language outside of the pattern matcher.

The IS statement is used to retrieve information from a data base by instantiating a pattern. If the instantiation fails, then, unless explicitly prohibited, the instantiation is attempted again using the rules of inference specified or any rules of inference available in the context and domains searched.

The ACHIEVE statement is similar except that if both the search and inference are insufficient to instantiate the pattern, then the action specified, or any available actions, are used to try to achieve an instantiated pattern.

The ASSERT and REMOVE statements are used to add and delete tuples from a context or domain. In each case, unless specifically prohibited, the consistency of the data base is checked after the statement is executed. If an inconsistency is found, then the statement fails and the changes are undone.

The PERFORM statement behaves exactly like the IS statement except that if the pattern is instantiated, it is then evaluated. Finally, the FAIL statement is used to explicitly invoke the fail mechanisms described earlier.

EXAMPLE

The following annotated example of the system's planned behavior was derived from one in the QLISP Manual[9]. The original problem statement is:

To make people happy either find a compatible marriage or make them rich. A marriage is compatible if both people are unmarried, of opposite sex, have a hobby in common and the wife is not more than five years older than the man. Someone is rich if their net worth is over a million dollars.

After engaging in a dialogue with the user (suppressed here), the system would arrive at the Loose Model stage in which the following informal description of the problem and domain exists:

1. (AKO PERSON OBJECT)
 2. (ARO SEX PERSON (ONE-OF MALE FEMALE))
 3. (ARO MARITAL-STATUS PERSON (ONE-OF MARRIED UNMARRIED))
 4. (ARO EMOTION PERSON (ONE-OF HAPPY SAD BLAH))
 5. (ARO NETWORTH PERSON NUMBER)
 6. (ARO WEALTH PERSON (ONE-OF RICH MIDDLE POOR))
 7. (ARO HOBBIES PERSON (SET ACTIVITY))
 8. (AKO BELLY-DANCING ACTIVITY)
 9. (AKO GARDENING ACTIVITY)
 10. (AKO PROGRAMMING ACTIVITY)
 11. (ARO AGE PERSON NUMBER)
 12. (ACTION MARRY (PARAMETERS PERSON PERSON#1)
 (CONTROLLED-BY SYSTEM)
 (PRECONDITIONS (AND (UNMARRIED PERSON#1)
 (UNMARRIED PERSON)
 (NEQ (SEX PERSON#1)
 (SEX PERSON))))
 (DESCRIPTION (ASSERT (MARRIED PERSON#1 PERSON)))
 (POSTCONDITIONS (MARRIED PERSON#1 PERSON)))
 13. (CONSTRAINT (PERSON#1 PERSON#2 PERSON#3)
 (MARRIED PERSON#1 PERSON#2)
 (MARRIED PERSON#1 PERSON#3))
 14. (CONSTRAINT (PERSON) (MARRIED PERSON PERSON))
 15. (IMPLIES (PERSON#1 PERSON#2) (MARRIED PERSON#1 PERSON#2)
 (MARRIED PERSON#2 PERSON#1))
 16. (IMPLIES (PERSON) (GT (NETWORTH PERSON) 1000000)
 (PERSON RICH))
 17. (AKO WIFE PERSON FEMALE MARRIED)
 18. (AKO HUSBAND PERSON MALE MARRIED)
 19. (ACTION MAKEHAPPY (PARAMETERS PERSON)
 (CONTROLLED-BY USER)
 (DESCRIPTION MAKEHAPPY (IF (OR (PERSON RICH)
 (HAS PERSON COMPATIBLE-MARRIAGE))
 (ASSERT (PERSON HAPPY)))))
 20. (AKO MARRIAGE EVENT MARRY)
 21. (AKO (COM.ATIBLE-MARRIAGE) MARRIAGE
 (LT (AGE WIFE) (AGE HUSBAND+5))
 (EXISTS (HOBBY#1) (AND
 (HOBBY HUSBAND HOBBY#1)
 (HOBBY WIFE HOBBY#1))))

The impression to be gained from the Loose Model stage is that the informal description is closely related to the natural language input given the system. The major problems of understanding this representation and transforming it into an operational program are left for the loose to precise translation.

Some of the items above are imprecise and are modified as part of model completion. For example, Item 17, above, must be changed to (AKO WIFE PERSON (SEX * FEMALE) (MARITAL-STATUS * MARRIED)).

Many other transformations are needed to arrive at the Precise Model program given below, only two of which will be dealt with here. The first occurs in Item 21, above, which attempts to find a hobby in common between the husband and wife. As written, it attempts to find a hobby which is the value of the HOBBY relation on husbands and wives. Realization that husbands and wives are persons and thus, that the HOBBY relation is well-defined in that regard, occurs automatically within the typing mechanism of the system. However, in attempting to find the common hobby, it must be noticed that only activities can be the value of HOBBY. Hence this pattern must be rewritten to look for an activity which is in common between the husband and wife.

More indicative of the types of problems encountered in the translation process are the mechanisms involved in the interpretation of (HAS PERSON COMPATIBLE-MARRIAGE) in Item 19. The system starts by seeing if PERSON and COMPATIBLE-MARRIAGE are related by the HAS relation. They are not. Now the system knows (!) that "HAS" is used very sloppily in English, so it looks to see how PERSON and COMPATIBLE-MARRIAGE are related. COMPATIBLE-MARRIAGE is A-KIND-OF marriage and MARRIAGE is A-KIND-OF the event of marrying. MARRIAGE is an action involving two persons; even more, it asserts that the two are related by the MARRIED relation. Hence, if the relation between "MARRIAGE" and "MARRIED" is linguistically known, the system assumes that "HAS MARRIAGE" is the same as "IS MARRIED". Notice that MARRIED is being used in two ways: first, as an attribute value of marital status, and second, as a relation between two people. In fact, the marital status is being further specified by whom the marriage is with. Finally, there is the issue of when the condition for compatible marriage is applicable: When the marriage occurred or when the question was asked? In addition, notice that within Item 21, above, that wife and husband are not existentially quantified but relate to the partners in the marriage. Thus, from the inferred fact that the person is married, the system must pick up the partner and use that pair to bind the husband and wife by type constraints in evaluating this condition. The result of this expansion is shown in the MAKE-HAPPY function below. All in all, the chain of processing required in the loose to precise translation is rather complex and ill-defined.

```

(MAKEHAPPY
  [LAMBDA (PERSON)
    (PROG (PERSON#1 PERSON#2 PERSON#3 ACTIVITY)
      (ACHIEVE
        (OR (WEALTH PERSON RICH)
            (AND (MARRIED PERSON PERSON#1)
                [LT
                  (PLUS [AGE (PERSON#2 (SEX * MALE)
                    (IN (ONE-OF PERSON PERSON#1)
                      5)
                    (AGE (PERSON#3 (SEX * FEMALE)
                      (IN (ONE-OF PERSON PERSON#1)
                        (HOBBY PERSON ACTIVITY)
                        (HOBBY PERSON#1 ACTIVITY)))
                  THEN (ASSERT (EMOTION PERSON HAPPY)))]))

(MARRY
  [LAMBDA (PERSON PERSON#1)
    (PROG (PERSON#2 PERSON#3)
      (CONSTRAIN (NOT (MARRIED PERSON PERSON#2))
        (NOT (MARRIED PERSON#1 PERSON#3))
        (NEQ (SEX PERSON#1)
            (SEX PERSON)))
      (ASSERT (MARRIED PERSON PERSON#1)))]))

(CONSTRAINT0001
  [LAMBDA (PERSON)
    (PROG (PERSON#1 PERSON#2)
      (NOT (AND (MARRIED PERSON PERSON#1)
        (MARRIED PERSON PERSON#2)))]))

(CONSTRAINT0002
  [LAMBDA (PERSON)
    (PROG NIL
      (NOT (MARRIED PERSON PERSON)))]))

(INFERENCE0001
  [LAMBDA (PERSON PERSON#1)
    (PROG NIL
      (IS (MARRIED PERSON PERSON#1)
        (THEN (ASSERT (MARRIED PERSON#1 PERSON)))
        (ELSE)))]))

(INFERENCE0002
  [LAMBDA (PERSON)
    (PROG NIL
      (IS (GT (NUMBER (NETWORTH PERSON *))
        1000000)
        (THEN (ASSERT (WEALTH PERSON RICH)))
        (ELSE)))]))

```

CONCLUSION

Although a wealth of problems remain unsolved (and undiscovered), a clear direction has been established. Domain-independent automatic programming has been divided into two parts: dialogue-driven acquisition of the domain semantics, and translation of ill-defined specifications into a precise form. Work is focusing on creating a core system for experimentation and on explicating the transformations in the Domain Acquisition and Model Completion modules. The implementation has been started and the Interpreter, Data Base Handler, and Precise Model form are all well in hand. An initial knowledge representation has been selected.

Despite the early stage of the project, several technical contributions have emerged in addition to the overall approach outlined above. AP/1 supports the intermixing of patterns to be instantiated and expressions to be evaluated. This greatly simplifies program control structure by obviating the need for explicit low-level search and control mechanisms. Knowledge has been highly structured through the strong use of types and the use of constraints on the arguments of tuples. Finally, techniques have been described for converting constraints and inferences, as well as actions, into procedures and for using procedures in both a goal-directed and applicative manner.

REFERENCES

- 1 Balzer, R. M., Automatic Programming, USC/Information Sciences Institute RR-73-1, September 1972, (draft).
- 2 Balzer, R. M., "A Global View of Automatic Programming," Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford University, August 20-23, 1973, pp. 494-499.
- 3 Project MAC Progress Report X, July 1972-July 1973, The Massachusetts Institute of Technology, Cambridge, Mass., 1973, pp. 174-176.
- 4 Fillmore, C. J., "The Case for Case", in Universals and Linguistic Theory, E. Bach and R. T. Harms (eds.), Holt, Rinehart, and Winston, 1968, pp. 1-88.
- 5 Balzer, R. M., Human Use of World Knowledge, USC/Information Sciences Institute, RR-73-7, March 1974.
- 6 Balzer, R. M., AP/I - A Language for Automatic Programming, USC/Information Sciences Institute, RR-73-13 (In progress).
- 7 Teitelman, W., D. G. Bobrow, A. K. Hartley, and D. L. Murphy, BBN-LISP TENEX Reference Manual, Bolt Beranek and Newman Inc., July 1971.
- 8 McDermott, D. V. and G. J. Sussman, Son of Conniver, The Conniver Reference Manual, Version II, The Massachusetts Institute of Technology, Cambridge, Mass., 1972.
- 9 Reboh, R., and E. Sacerdoti, A Preliminary OLISP Manual, Stanford Research Institute, Artificial Intelligence Center, Technical Note 81, August 1973.